Learning From Thousands of Build Failures of Linux Kernel Configurations

Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Djamel Eddine Khelladi, Jean-Marc Jézéquel

Learning From Thousands of Build Failures of Linux Kernel Configurations

Mathieu Acher*, Hugo Martin*, Juliana Alves Pereira*, Arnaud Blouin*, Djamel Eddine Khelladi†, Jean-Marc Jézéquel*
*Univ Rennes, Inria, CNRS, IRISA, Rennes, France. Emails: firstname.lastname@irisa.fr
†CNRS, IRISA, Rennes, France. Email: firstname.lastname@irisa.fr

***Abstract*—The Linux kernel offers more than ten thousands configuration options that can be combined to build an almost infinite number of kernel variants. Developers and contributors spend significant effort and computational resources to continuously track and hopefully fix configurations that lead to build failures. In this experience paper, we report on our endeavor to develop an infrastructure, called TUXML, able to build any kernel configuration and learn what could explain or even prevent configurations' failures. Our results over 95,000+ configurations show that TUXML can accurately cluster 3,600+ failures, automatically trace the responsible configuration options, and learn by itself to avoid unnecessary and costly builds. Our large qualitative and quantitative analysis reveals insights about Linux itself (e.g., we only found 16 configuration bugs) and the difficulty to engineer a build infrastructure for configurable systems (e.g., a false positive failure may mask true configuration bugs).**

***Index Terms*—configurable systems, software testing, software product lines, operating systems, Linux kernel, build systems**

I. INTRODUCTION

Linux is one of the most complex configurable system ever developed with more than 15,000 configuration options for recent versions of the kernel. Users can activate options – either compiled as modules or directly integrated within the kernel – and deactivate options. When all options are combined together and form a *configuration*, a kernel can be fully compiled and built. Though there are some logical constraints among options, the number of configurations is almost infinite and so the number of possible kernel variants. An intriguing question is whether and how much variants of the Linux kernel build. A correct build is a mandatory prerequisite before investigating whether the kernel boots, passes test suites, *etc.*

In practice, developers and contributors spend significant effort and computational resources to continuously track configurations that lead to build failures. While validating the correctness of the kernel across its entire configuration space is desirable, exhaustive testing of all configurations is simply impossible at the scale of Linux. Furthermore, the build process of Linux involves different layers and languages (CPP, Make and Kconfig) that are hard to statically analyze. Though formal methods and program analysis can identify some classes of defects [1], [2] – leading to variability-aware testing approaches (*e.g.,* [3]–[5]) – a common practice is still to build a sample of (representative) configurations.

In complement to static analysis, there are some initiatives, like 0-day [6], [7], to compile and build kernels out of either default configurations (*e.g.,* tinyconfig), users' custom configurations, or simply random configurations. Build failures may occur, may be reported on mailing lists, and may eventually be fixed. In this work we aim to provide a build infrastructure for automatically exploring the configuration space of the Linux kernel and assist developers in finding problems as early as possible. A general and central issue is to analyze and continuously manage *build failures caused by specific configurations*; our idea is to leverage learning techniques for understanding large configuration data and piloting the build infrastructure. There are several related challenges.

First, the infrastructure should be as *sound* as possible and avoid false compilation failures that are due to the built environment itself. Prior to the building, many tools and packages should be installed and only pop out when specific options or combinations of options are activated. Even advanced users can be fooled by the documentation or the build process. Second, there is the need to *understand* build failures: Is the failure due to the built environment or to an actual bug in the Linux source code? What (combinations of) options are responsible for the failure? There is the risk that builds failures propagate to many other configurations and cause a significant waste of resources (the build process takes 15 minutes on average on a recent machine). Furthermore, configuration failures can mask some other failures – this phenomenon is not well-understood and our goal is to better characterize and quantify it with the study of Linux. In an ideal case, developers of TUXML or Linux can immediately understand and fix configuration failures. Unfortunately, the fixing process and its adoption take effort and time. Instead of waiting for a hypothetical patch, the infrastructure could predict that some combinations of options will cause a fail. Hence, a third requirement is that the build infrastructure should be able to *prevent* as early as possible unnecessary builds of some configurations. We expect that TUXML is capable of taking such decisions without a human in the loop. Of course, developers are then extremely useful to understand the precise cause of failures and patch the kernel.

Builds' errors are subject to intensive research [8]–[11] and this work specifically considers configurations' builds of the same system. Several studies report that configuration failures are expensive, common in both open source and commercial software systems, and represent one of the most common types of failures [12], [13]. There are many approaches that

aim to efficiently cover configurations' failures or bugs [14]–[20]. The Linux kernel is an industry-strength case for further understanding the phenomenon of build failures within an enormous configuration space. A few empirical studies [21]–[27] have considered relevant aspects of Linux (build system, variability implementation, constraints, bugs, compilation warnings). However, these studies did not concretely build configurations in the large to observe potential failures.

This paper first describes an infrastructure, called TUXML, able to build any kernel configuration and learn what could explain or even prevent configurations' failures. We show that the sole use of statistical learning or failures' errors clustering have limitations, and the key resides in *combining* the two learning techniques to identify what causes a failure or a set of failures. Our results over 95,854 configurations show that TUXML can accurately group together 3,600+ failures, automatically trace the responsible configuration options, and quickly learn by itself to avoid unnecessary and costly builds. Our large-scale analysis also reveals qualitative insights about configurations' failures (*e.g.,* masking effects) and the difficulty to engineer a build infrastructure for configurable systems.

The contributions of this paper are as follows:

- TUXML an open-source, learning-based infrastructure for building Linux kernel configurations in the large [28];
- Learning techniques to cluster failures' errors and map them to faulty (combinations of) options with two applications: configuration bug understanding and automated prevention of build failures;
- A characterization of configuration phenomena not well understood, namely: configurations' bugs interactions, masked failures, masked configuration bugs, and bugs dominance;
- A quantitative and qualitative analysis of configurations' build failures, bugs and fixes of Linux;
- A comprehensive dataset of builds and failures for replication and reproducibility of the results [28].

ABC of SE [29]. Our experience paper mainly contributes to a knowledge-seeking study. Specifically, we perform a field study of Linux, a highly-configurable system and mature project. We gain insights about the phenomenon of configurations' build failures using a very large corpus (95K+ configurations). Our contribution is also a solution-seeking study since we combine learning techniques to classify, predict, and prevent failures.

Audience. The intended audience of this paper includes but is not limited to Linux contributors. Researchers and practitioners in configurable systems shall benefit from our analysis of configuration failures. We also believe developers and operationals in charge of engineering or using build infrastructure can benefit from our experience report.

II. BACKGROUND AND TERMINOLOGY

A. Linux, options, and configurations

The Linux kernel is a prominent example of a highly-configurable system. Thousands of options (also called features) are available on different architectures (*e.g.,* x86, amd64, arm) and documented in several Kconfig files. For the architecture x86 and for the version 4.13.3, developers can use 12,659 options to tailor functional and non-functional needs of a particular use (*e.g.,* embedded system development). The majority of options has either boolean values ('y' for activating an option or 'n' for deactivating an option) or tri-state values ('y', 'n', and 'm' for activating an option as a module). There are also numerical or string values while default values can also be specified. Since not all combinations of values are possible, there are numerous constraints between options. For instance, the option `DRM_VBOXVIDEO` depended on `CONFIG_DRM`, `CONFIG_X86`, `CONFIG_PCI`, and `DRM_KMS_HELPER` in the version 4.13.3 (see Figure 1). Users of the Linux kernel set values to options (*e.g.,* through a configurator [30]) and obtain a so-called .config file. We consider that a *configuration* is an assignment of values to all options.

```
@@ -2,6 +2,8 @@ config DRM_VBOXVIDEO
         tristate "Virtual Box Graphics Card"
         depends on DRM && X86 && PCI
         select DRM_KMS_HELPER
+        select GENERIC_ALLOCATOR
+        select DRM_TTM
         help
           This is a KMS driver for the virtual
            ↪  Graphics Card used in
           Virtual Box virtual machines.
```

Figure 1: Option DRM_VBOXVIDEO (2 patches, version 4.14)

Based on a configuration, the build process of a Linux kernel can start and involves different layers, tools, and languages (C, CPP, gcc, GnuMake and Kconfig). The conditional compilation directives of the C preprocessor (CPP) are intensively used to implement options. CPP directives, like #ifdef and #endif, enclose the variable code that can be included or excluded for different configurations. Scattered options are spread over different files of the code base, possibly across subsystems [31]. The complexity of the build process, the scattering of options in the source code, and the enormous configuration space challenge Linux developers [21]–[25], [27]. A missing CPP directive in a file can typically lead to a build failure.

B. Configuration failure and configuration bug

A *failure* is an "undesired effect observed in the system's delivered service" [32], [33]. In the context of configurable systems, a failure may occur due to specific options' values (configurations). In the rest of the paper, we use the term failure to refer to a *configuration failure*. In particular, we are interested in Linux kernel configurations that fail to compile and build. A failure can be logged and observed through diagnostic messages about build errors (see right-hand side of Figure 2).

We consider that a *bug* (also called *defect* or *fault*) is a cause of failures. A single fault can explain many configuration failures since the same combinations of options cause an undesired effect. For instance, a missing dependency between two options (see Figure 1) can be the reasons why many failures are observed. We use the term bug to refer to a *configuration bug*. Previous work on configurable systems made similar distinctions between failures and bugs [14], [17].

```
X86_64=y
USB=m
...
# always leads to a failure
AIC7XXX_BUILD_FIRMWARE=y
...
# always leads to a failure
DRM_VBOXVIDEO=y
GENERIC_ALLOCATOR=n
```

```
aicasm_symbol.c:48:19: fatal error: aicdb.h: No such
 ↪  file or directory
 #include "aicdb.h"
                   ^
make[4]: *** [aicasm] Error 1
make[3]: *** [drivers/scsi/aic7xxx/aicasm/aicasm]
 ↪  Error 2
make[2]: *** [drivers/scsi/aic7xxx] Error 2
make: *** [drivers] Error 2
```

(a) Configuration failure due to `AIC7XXX_BUILD_FIRMWARE`

```
X86_64=y
USB=y
...
AIC7XXX_BUILD_FIRMWARE=n
...
# always leads to a failure
DRM_VBOXVIDEO=y
GENERIC_ALLOCATOR=n
```

```
drivers/staging/vboxvideo/vbox_hgsmi.o: In function
 ↪  hgsmi_buffer_free': vbox_hgsmi.c:(.text+0x105):
 ↪  undefined reference to gen_pool_free'
drivers/staging/vboxvideo/vbox_main.o: In function
 ↪  vbox_hw_fini': vbox_main.c:(.text+0x14a):
 ↪  undefined reference to gen_pool_destroy'
drivers/staging/vboxvideo/vbox_main.o: In function
 ↪  vbox_driver_load': vbox_main.c:(.text+0x561):
 ↪  undefined reference to gen_pool_create'
vbox_main.c:(.text+0x58e): undefined reference to
 ↪  gen_pool_add_virt' vbox_main.c:(.text+0x59e):
 ↪  undefined reference to gen_pool_destroy'
make: *** [vmlinux] Error 1
```

(b) configuration failure due to `DRM_VBOXVIDEO`, `GENERIC_ALLOCATOR`

```
X86_64=y
# always leads to a failure
AIC7XXX_BUILD_FIRMWARE=y
# always leads to a failure
FORTIFY_SOURCE=y
UBSAN_NULL=y
UBSAN_SANITIZE_ALL=y
IPV6=y
INFINIBAND_ADDR_TRANS=y
...
```

```
./include/linux/string.h:305:4: error: call to
 ↪  '__read_overflow2' declared with attribute
 ↪  error: detected read beyond size of object
 ↪  passed as 2nd parameter
   __read_overflow2();
   ^~~~~~~~~~~~~~~~~~
make[3]: ***
 ↪  [drivers/infiniband/core/roce_gid_mgmt.o] Error
 ↪  1
make[2]: *** [drivers/infiniband/core] Error 2
make[1]: *** [drivers/infiniband] Error 2
make: *** [drivers] Error 2
```

(c) Configuration failure due to `FORTIFY_SOURCE` +options in red

Figure 2: Configuration failures and error messages. Which options' values cause the failures? How to prevent failures?

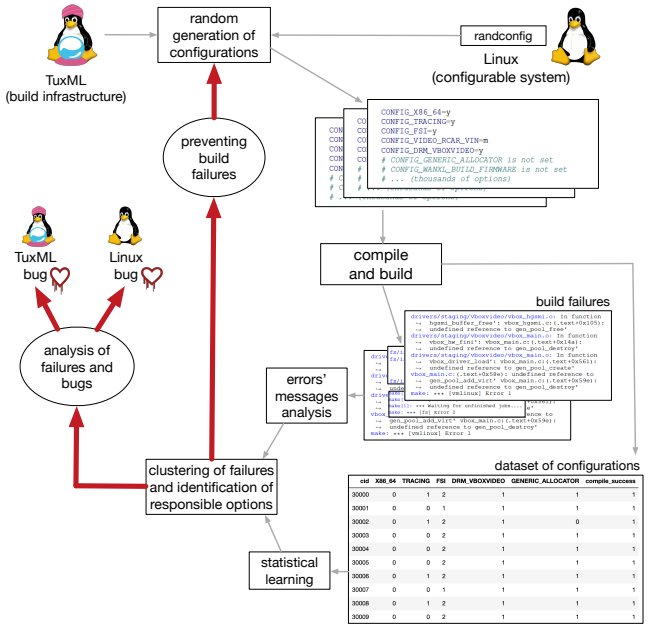


Figure 3: TuxML infrastructure for building configurations and learning from failures.

III. TuxML: A Learning-Based Build Infrastructure

Figure 3 presents an overview of the TuxML ideal process for building configurations. The infrastructure actively learns from a sample of randomly generated configuration failures to specialize the generation of additional configurations and prevent future build failures.

A. Implementation details of TuxML

We designed TuxML to build the Linux kernel in the large *i.e.,* whatever options are combined. Though a few basic tools might be sufficient for compiling a given configuration, more specific utilities are sometimes required due to the activation of some specific options. Our early experiments point up the need to pre-install many libraries to support the build of *any* configuration.

TuxML relies on Docker to host the numerous packages needed to compile the Linux kernel. Docker offers a reproducible and portable environment – clusters of heterogeneous machines can be used while contributors can participate in the collection of build data. Inside Docker, a collection of Python scripts automates the build process. An important step is the selection of configurations to build. We rely on randconfig to randomly generate Linux kernel configurations (see top of Figure 3). randconfig has the merit of generating valid configurations that respect the numerous constraints between options. It is also a mature tool that the Linux project develops and uses. Though randconfig is not producing uniform, random samples (see Section VI), there is a diversity within the values of options (being 'y', 'n', or 'm'). Given .config files, TuxML builds numerous kernels. Throughout the process, TuxML collects various kinds of information, including the final compilation status and the diagnostic messages of gcc, make, *etc.* We populate a database of configurations for further analyses (see bottom of Figure 3). Though this study focuses on the Linux version 4.13.3, TuxML supports all the kernel architectures and 4.0 versions.

B. Learning faulty options from failures

We quickly noticed that several configuration build failures may occur. In addition to the support of the build process, TuxML has learning mechanisms to automatically *understand* and *prevent* failures. There are several scenarios that involve three different actors: the TuxML maintainers, Linux developers, and TuxML itself as an *autonomous* service. In a first scenario, a failure is due to the TuxML environment *e.g.,* a missing tool or a programming error. TuxML should properly identify and handle such false positive failures. As shown in Figure 3, a first task is to make the distinction between false positive failures (caused by TuxML) and true positive failures (caused by Linux). Figure 2a reports a failure due to the `AIC7XXX_BUILD_FIRMWARE` option. As detailed in Section V, an in-depth analysis shows that TuxML is responsible for this failure. A second important task is to avoid false positive failures of the build infrastructure. It can be through bug fixing (if any) or simply through the deactivation of responsible options' values. The manual reviewing effort is labor-intensive when a large number of failures arise. For

both tasks, TUXML developers need support to understand the cause of failures.

In a second scenario, a failure is due to an actual configuration bug of Linux. Figure 2b shows a failure due to a missing dependency between `DRM_VBOXVIDEO` and `GENERIC_ALLOCATOR` (see the bug fix in Figure 1). Despite the understanding of what causes failures, a fix might not be available. In addition to the report and confirmation of the Linux bug, developers of TUXML can prevent the infrastructure to build configurations involving faulty (combinations of) options. For example, TUXML can be specialized to generate random configurations without `DRM_VBOXVIDEO=y` and `GENERIC_ALLOCATOR=n`. Technically, there are two ways to specialize the generation of configurations: a priori through the preset of some individual options values before calling randconfig; a posteriori by eliminating generated .config files that contain the faulty combinations of options' values; we use both mechanisms.

The two previous scenarios involve humans to qualitatively assess failures and then decide how to react. TUXML can hardly decide whether it is a bug of Linux or a bug of the environment. It is also challenging to correct a configuration bug (*e.g.,* in the source code). Hence, in a third scenario, TUXML automatically prevents failures by avoiding the build of configurations with faulty options' values. It can be seen as an automated fix at the configuration level. It has the merit of saving precious resources until developers apply a real fix of TUXML or Linux.

IV. LEARNING FROM CONFIGURATIONS' FAILURES

This section presents techniques to learn from configurations' builds and failures. We also introduce phenomena (*e.g.,* bugs interactions and masked failures) that hinder the automated analysis of Linux failures.

Let us consider the failure of Figure 2a. The problem is to automatically find, among thousands of options, which combinations cause the failure. Just looking at the diagnostic message is not sufficient to precisely locate faulty options. There are dozens of C files and options involved in the realization of the `aic7xxx` driver.

A. Statistical learning and bugs' interactions

Our strategy is to use statistical, supervised learning to predict whether configurations fail. The principle is to learn out of a sample of configuration data. The hope is to generalize to the rest of the data and to make an accurate prediction of configurations that have not been built. TUXML can typically use a classifier to prevent the build of configurations that will fail. Many classification algorithms exist and some are able to report what combinations of options lead to a failure. We chose the scikit-learn implementation of Classification and Regression Trees (CART). CART recursively partitions the training sample: At each step, a variable (*i.e.,* an option) that best splits the set of configurations is chosen and CART determines which split minimizes the entropy or the Gini index of the class distribution (here, the class is binary: "failure" or "success"). Our experiments showed that entropy is best suited to our case. Rules can be extracted through the traversal of the tree until reaching the leaf nodes that predict the outcome. All edges are connected by logical `AND` and information about options values. An example is given in Figure 4. There are four leaf nodes leading to a build failure. The first rule is `AIC7XXX_BUILD_FIRMWARE=y` stating that the sole activation of this option fails. It explains why the configuration of Figure 2a fails. The fourth rule is more complex: `AIC7XXX_BUILD_FIRMWARE=n` $\wedge$ `AIC79XXX_BUILD_FIRMWARE=n` $\wedge$ `WANXL_BUILD_FIRMWARE=n` $\wedge$ `GENERIC_ALLOCATOR=n` $\wedge$ `DRM_VBOXVIDEO=y`. It explains why the configuration of Figure 2b fails.

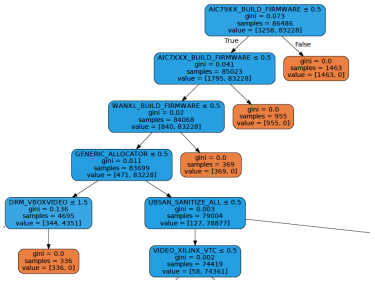


Figure 4: Decision tree (excerpt, training: 10% of the dataset)

Despite accurate and actionable results, the decision tree of Figure 4 has limitations.

Masked bug. The fourth rule suggests that `GENERIC_ALLOCATOR=n, DRM_VBOXVIDEO=y` lead to a failure *under the condition* other specific options' values are set (*e.g.,* `AIC7XXX_BUILD_FIRMWARE=n`). It is actually misleading and an unnecessary condition: An analysis over the whole dataset shows that `GENERIC_ALLOCATOR=n, DRM_VBOXVIDEO=y` *always* lead to a failure, whatever are the other values (even when `AIC7XXX_BUILD_FIRMWARE=y` holds). It is no surprise looking at the fix of Figure 1 and configuration failures of Figures 2a and 2b. Hence, an unintended effect is that rules of the decision tree may mask the presence of some faulty options, i.e. the decision tree path is not reliable. In particular, `GENERIC_ALLOCATOR=n, DRM_VBOXVIDEO=y` could well be the cause of the failure of Figure 2a. Consequently, statistical-based rules can also wrongly explain a failure. Another example is given in Figure 2c: here two bugs are present in the configuration. According to the decision tree, `AIC7XXX_BUILD_FIRMWARE` is the cause of the failure. However, the error message is actually due to the combination of `FORTIFY_SOURCE` with other options in red.

Summary and implications. When there are several bugs in a configuration (what we call hereafter *bugs interactions*), a configuration bug masks other bugs. The effect of masked bugs is twofold w.r.t. configuration failures: (1) in terms of understanding, developers may be exposed to misleading explanations; (2) TUXML or developers can miss an opportunity to prevent failures.

Masked failure. The decision tree considers that the failure

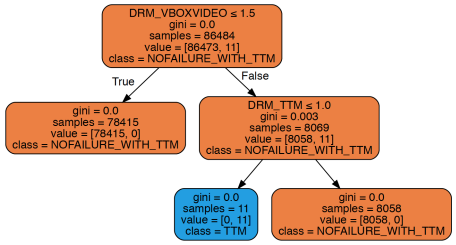


Figure 5: The full decision tree: per-cluster classification over TTM-like failures' messages

of Figure 2a is due to `AIC7XXX_BUILD_FIRMWARE=y`. Fortunately, the statistical-based explanation is a good one, since the error message is in line with its cause. However, an error message like the one of Figure 2b could have arisen since `GENERIC_ALLOCATOR=n`, `DRM_VBOXVIDEO=y` hold in Figure 2a. This example shows an important phenomenon: the failure of `GENERIC_ALLOCATOR=n`, `DRM_VBOXVIDEO=y` is not observable when `AIC7XXX_BUILD_FIRMWARE=y` holds (which we call *bug dominance*). Bug dominance is another orthogonal concept that holds when a configuration bug (says $cb1$) dominates other bugs, i.e. when failures related to $cb1$ always pop out first and mask other failures. In the example of Figure 2c, the bug `FORTIFY_SOURCE` (with other options in red) dominates the bug `AIC7XXX_BUILD_FIRMWARE`.

Summary and implications. When there are bugs interactions within a configuration, a configuration failure masks the manifestation of other failures. The effect of masked failures is twofold w.r.t. configuration bugs: (1) in terms of understanding, developers may miss an opportunity to find a bug; (2) TuxML or developers can miss an opportunity to prevent failures and spend many resources with a dominant bug.

B. Clustering errors' messages to the rescue

To mitigate the effects of bugs' interactions, our idea is to exploit failures' messages. A first step is to group together similar configurations failures based on errors' messages. The intuition is that configurations with the same or similar errors' messages have more chance to have a similar cause. We automatically build clusters of errors based on some pre-defined patterns (see our experience in the next section).

Then, we label each configuration with an error cluster. Thus, we address the problem of classifying configurations in terms of error cluster instead of simply predicting whether the build fails. Compared to the previous approach, we now resolve a *multiclass* classification. We can reuse CART algorithms. This time, the tree explicitly points out how a set of similar error messages (failures) connects to faulty options. Developers of TuxML can trace, for a given error message (cluster), what are the responsible options. Hence, the masked bug and misleading explanation can be mitigated since the tree now reasons about the error message.

Following a similar approach, it is also possible to train a classifier *per specific class of error* (instead of considering all kinds of errors). An example is given in Figure 5: The classifier specifically predicts whether the TTM failure (a cluster of error) arises in a given configuration. Otherwise, it is either a build success or another kind of failure. The tree of Figure 5 is very concise and points out a potential Linux bug that involves `DRM_VBOXVIDEO` and `DRM_TTM` – something not showed in the decision tree of Figure 4. We give further insights about this bug in our study.

In the next section, we systematically provide quantitative and qualitative insights about the phenomena. We also report on the strengths and limits of learning techniques for the task of understanding failures and bugs – we eat our own food.

V. Study

The implementation of TuxML has been initiated in September 2017. The goal of this section is to report our TuxML experience in applying automated techniques to (1) build Linux kernel configurations at large-scale; (2) learn, understand, and prevent failures.

Scope. Since the beginning, we chose to only focus on the kernel version 4.13.3 (release date: 20 Sep 2017). Furthermore, we specifically target the x86-64 architecture *i.e.,* technically, all configurations have values `CONFIG_X86=y` and `CONFIG_X86_64=y`.

Datasets. We made hundreds of builds before reaching a certain maturity of TuxML. In June 2018, we considered that TuxML was ready for a large-scale experiment. We used a cluster of 80 machines to build 95,854 random configurations. We can distinguish two phases:

- a random generation for the first $\approx$ 60K configurations;
- for the remaining configurations, a specialized TuxML that prevents failures and precludes the use of three faulty options responsible to most of the previous failures.

In total, we invested 15K hours of computation time. A build took more than 9 minutes per configuration on average (standard deviation: 11 min). The number of options with more than one value is 9,286 *i.e.,* there are nine thousand predictors that can potentially explain a build success or a failure.

The main research question is: *How do* TuxML *developers deal with configuration failures?* It involves the qualitative and quantitative assessment of the following activities:

- **(A1) finding and characterizing failures:** How many different configuration failures do TuxML developers found? How many configuration failures are due to TuxML bugs or to Linux bugs?
- **(A2) understanding process:** How automated techniques help to understand the cause of failures? To what extent bugs' interactions hinder the analysis process?
- **(A3) preventing failures:** Can TuxML autonomously prevent failures? What is the amount of builds needed to accurately prevent failures?

Methodology. The analysis of failures is based on automated techniques and manual investigation. All co-authors of this paper participate in the analysis. Some were not directly involved in the development of TuxML. For all failures, we

systematically seek to understand the possible bugs, how the Linux community has fixed them (*e.g.,* through the analysis of mailing lists or commits), and whether the fix has been effective for further versions of Linux (*e.g.,* version 4.14). We split up the work to have different perspectives and approaches over the large configuration data. We had regular meetings to discuss our findings and report on our difficulties.

A. (A1) Finding and characterizing configurations' failures

How many configuration failures did we find? In total, we observed 3,614 configurations failures out of 95,854 configurations. Without specialization, we collected 3,460 failures out of 59,366 configurations. That is, 5.83% of build lead to failures. As shown in Table I, three faulty options `AIC7XXX_BUILD_FIRMWARE`, `AIC79XXX_BUILD_FIRMWARE`, `WANXL_BUILD_FIRMWARE` explain 5.22% of failures. Without them, 359 configurations fail (0.61%). During the specialization of TuxML (phase 2 of the experiment), we deactivate these faulty options. The prevention of failures was effective: we only observed 161 failures out of 36,729 – 0.44% fail.

To what extent failures' messages differ? Diagnostic messages of gcc and make are verbose, full of technical terms, noise, and long (the median of the number of lines is 28). As in [25], we found many warnings (92% of failures contain warning messages). As it is not possible to individually review thousands of failures, an approach is to group together similar messages. On the one hand, we quickly noticed that a fully unsupervised clustering produces too many clusters, most being unexploitable. On the other hand, very specific patterns of errors, based on manual observations, can be incomplete and we miss some failures.

We made two attempts with similar results. The first one was to find generic patterns about errors like "undefined references". We relied on top frequent terms to find other patterns and cover all failures. Based on k-means clustering, we obtained 33 clusters. After a manual review, we eventually merged together some clusters. The second attempt was based on a manual, iterative specification of patterns until covering all failures. In both cases, we found that a fully automated clustering is not desirable and human supervision is beneficial to control the granularity of the information and not to miss important cases. It also allows us to find generic and reusable patterns of errors. The result of this process are clusters reported in Table I. There are 19 patterns out of which 4 covers 95% of the failures. There are also very rare occurrences with 5 clusters appearing only once.

How many configuration failures are due to TuxML *bugs or to Linux bugs?* For each cluster of failures, there is a need to identify the cause and distinguish whether it is an issue of the build infrastructure. Only looking at the diagnostic messages has limitations: We also need to understand the configuration context in which the failure occurs.

For instance, the decision tree in charge of classifying builds as failure or success (binary classification) pointed out the influence of options `AIC7XXX_BUILD_FIRMWARE` and `AIC79XX_BUILD_FIRMWARE`. These two options necessarily raise failures' errors `AICDB` or `AIC7XXX`. Moreover, the messages' content is related to the two options. The documentation of the option `AIC7XXX_BUILD_FIRMWARE` gives the explanation:

> This option should only be enabled if you are modifying the firmware source to the aic7xxx driver and wish to have the generated firmware include files updated during a normal kernel build. The assembler for the firmware requires lex and yacc or their equivalents, as well as the db v1 library. You may have to install additional packages or modify the assembler Makefile or the files it includes if your build environment is different than that of the author.

Bugs of TuxML. Looking at the error messages, configuration and documentation, `AICDB` and `AIC7XXX` are clearly a TuxML issue since some tools are missing. However the set up is not straightforward to instrument. A pragmatic decision could be to prevent the compilation of the firmware, instead of over-complexifying the build infrastructure. In fact, a further look at the documentation shows that there exists a dependent option of `AIC7XXX_BUILD_FIRMWARE` called `CONFIG_PREVENT_FIRMWARE_BUILD`. The documentation for the version 4.13.3 states:

> Say yes to avoid building firmware. Firmware is usually shipped with the driver and only when updating the firmware should a rebuild be made. If unsure, say Y here.

At first glance, `CONFIG_PREVENT_FIRMWARE_BUILD` is an atypical option: its only role is to prevent the compilation of other options – one may wonder why not simply putting 'n' to firmware options. The reason is that firmware options might actually be needed elsewhere: With this (old) mechanism, it is still possible to symbolically activate them, without actually building. Interestingly, the documentation of `CONFIG_PREVENT_FIRMWARE_BUILD` has been updated in version 4.18 with the following commit comment:

> Even the documentation for PREVENT_FIRMWARE_BUILD is a bit fuzzy and how it fits into this big picture.

The bug with `WANXL_FIRMWARE_BUILD` (see Table I) is very similar: specific tools like as68k are needed to build the option while the use of `CONFIG_PREVENT_FIRMWARE_BUILD` is possible. The documentation for the version 4.13.3 is as disturbing.

Overall, though `CONFIG_PREVENT_FIRMWARE_BUILD` was poorly documented and actually never activated in our dataset, we consider that three bugs are due to TuxML.

Key lessons learned. Specific and rare tools are sometimes required to build specific options: their inclusions in the build infrastructure are hard to anticipate. TuxML itself should be monitored/tested as a configurable system. Furthermore, configuration-dependent tools might be hard to find and deployed, up to the point there are some mechanisms to prevent the build of some options.

Bugs of Linux. We analyze other clusters of failures of Table I together with their configurations; we came to the conclusion that all these failures are due to a Linux bug.

cluster	error message	nb_failures	percentage	bug (faulty option)	Bug?	Fix
AICDB	aicdb.h: No such file or directory	2464	68.05	AIC7XXX_BUILD_FIRMWARE ! AIC79XX_BUILD_FIRMWARE	TuxML	missing tools / Kconfig doc.
AS68K	as68k: not found	476	13.15	WANXL_BUILD_FIRMWARE	TuxML	missing tools / Kconfig doc.
GEN	undefined reference to 'gen_pool	367	10.14	DRM_VBOXVIDEO & GENERIC_ALLOCATOR	Linux	Kconfig dependency
AIC7XXX	[drivers/scsi/aic7xxx/aicasm/aicasm] Error 2	161	4.45	AIC7XXX_BUILD_FIRMWARE ! AIC79XX_BUILD_FIRMWARE	TuxML	missing tools / Kconfig doc.
OVERFLOW2	__read_overflow2	83	2.29	FORTIFY_SOURCE & UBSAN_SANITIZE_ALL & INFINIBA...	Linux	source code
V4L2	undefined reference to 'v4l2	19	0.52	VIDEO_MUX & VIDEO_V4L2	Linux	Kconfig dependency
BLACKLIGHT	undefined reference to 'backlight_device	15	0.41	BACKLIGHT_CLASS_DEVICE & DRM_I915 ! DRM_SAVAGE...	Linux	Kconfig dependency
TTM	undefined reference to 'ttm	13	0.36	DRM_VBOXVIDEO & DRM_TTM	Linux	Kconfig dependency
CONFIG_NLS_DEFAULT	CONFIG_NLS_DEFAULT	6	0.17	NLS & ..	Linux	source code
DRM_BRIDGE_CLUSTER	undefined reference to 'drm_panel_bridge_add	3	0.08	SPI_JCORE & ..	Linux	source code
PINCTRL_CLUSTER	[drivers/pinctrl/pinctrl-mcp23s08.o] Error	3	0.08	GPIOLIB & ..	Linux	Kconfig dependency
CRC32_CLUSTER	undefined reference to 'crc32	2	0.06	CRC32 & VIDEO	Linux	Kconfig dependency
BTBCM	undefined reference to 'btbcm_set_bdaddr	2	0.06	BT_HCIUART_H4 & ..	Linux	Kconfig dependency
DEVM	undefined reference to '__devm_regmap	2	0.06	REGMAP_MMIO & ..	Linux	Kconfig dependency
PCM	undefined reference to 'atmel_pcm_dma_platform	1	0.03	VIDEO_SAA7134_GO7007 & SND_SOC_RT5514_SPI	Linux	Kconfig dependency
ULPI	undefined reference to 'ulpi	1	0.03	USB_F_TCM & ..	Linux	Kconfig dependency
BLACKLIGHT2	error: 'intel_backlight_device_register'	1	0.03	VIDEO_SOLO6X10 & ..	Linux	source code
I2C_CLUSTER	error: implicit declaration of function 'i2c_g...	1	0.03	VIDEO_ATOMISP & ..	Linux	source code + Kconfig dep.
DEVM2	undefined reference to 'devm_of_led	1	0.03	NEW_LEDS & ..	Linux	Kconfig dependency

Table I: Clusters of error messages, frequencies' failures, and faulty options responsible of the failure

Most of the bugs have been resolved through the edit of Kconfig files, typically to fix a missing dependency. Two prominent examples (cluster TTM vs cluster GEN) have already been given early in the paper: Figure 1, page 2 shows two patchs to resolve an issue with DRM_VBOXVIDEO. Two different contributors independently propose the two patchs. The manifestation of the failure as well as the fix of faulty combination of options differ. Some bugs have been corrected at the source code level. Figure 6 provides an example. At the Kconfig implementation level, a pre-condition between the option JOLIET and NLS was missing. More significant changes at the source code level are sometimes applied to fix bugs. There are also cases in which both Kconfig and the source code have been modified.

```
+#ifdef CONFIG_JOLIET
         if (sbi->s_nls_iocharset &&
         strcmp(sbi->s_nls_iocharset->charset,
          ↪  CONFIG_NLS_DEFAULT) != 0)
           seq_printf(m, ",iocharset=%s",
            ↪  sbi->s_nls_iocharset->charset);
+#endif
```

Figure 6: Option NLS (1 patch, version 4.14)

Impacts on bug prevention. Although bug fixes at the source code level are much harder to find and requires the involvement of Linux developers, fixes at the Kconfig level can theoretically be synthesized through statistical learning and the extraction of rules. For instance, we can infer that DRM_VBOXVIDEO depends on DRM_TTM (see the decision tree of Figure 5). In particular, we can have a preventive and temporary fix at the configuration level since we statistically observe that two options wrongly interact (*e.g.,* NLS with JOLIET).

Number of faulty options involved. Most of the bug fixes involve 2 options *e.g.,* a dependency is missing between two options. A noticeable exception is the bug with FORTIFY_SOURCE that involves 6 options participating to a novel compiler verification.

Community involvement. For all failures and bugs, we were able to find a related fix through mailing lists or commits. Some bugs have been detected by the robot of 0-day [6], [7] and others by contributors. All bugs have been fixed before version 4.17, the majority being fixed for version 4.14.

B. (A2) On the process of understanding failures

Table I results from an in-depth analysis of failures. We aim here to report on pitfalls, strengths and weaknesses of techniques to support the process. In short: **How automated techniques help to understand the cause of failures?**

Key lesson learned: bugs' interactions can be false positive interactions. In the binary classification, rules of the decision tree suggest unnecessary interactions between options. That is, the tree suggests a dependency between two options' values, but the dependency is simply pointless and incidental. In software engineering, the phenomenon of feature interaction is well-known and the cause of many failures: in general, many options depend on other options through inclusion or exclusion. But here the situation is different: options that interact can be both faulty. A representative example is the fourth rule of Figure 4: GENERIC_ALLOCATOR=n, DRM_VBOXVIDEO=y is sufficient to raise a failure, there is no need to have AIC7XXX_BUILD_FIRMWARE=n. From a statistical point of view, these rules make sense since accurate for predicting the outcome. However, correlation is not causation, and some rules are misleading. A general attitude is to make the hypothesis that a faulty (combination of options) may not truly interact with another bug. Hence, when reviewing failures and using the decision tree, we can forget such interactions and isolate the effects of other faulty options. However, this mental shift further complicates the analysis and is based on assumptions yet to be validated.

Key lesson learned: per cluster classification gives more concise and meaningful rules than binary classification.

In the binary classification setting, there are numerous rules (more than 60 rules with a training set size of 10% of the dataset) with dozen of options involved. Though it was extremely useful for top faulty options, we had practical difficulties to read the tree and identify responsible options of other failures. Arguably, such weaknesses can be due to the hyper-parameters of the decision tree (such as maximum depth or the measure of the splitting strategy): we experimented to set different parameters and we noticed similar difficulties.

A more profound reason is that some phenomena cannot be observed in the case we omit failures' messages: Only reasoning in terms of build success/failure masks the effects of some combinations of options. The way options are grouped

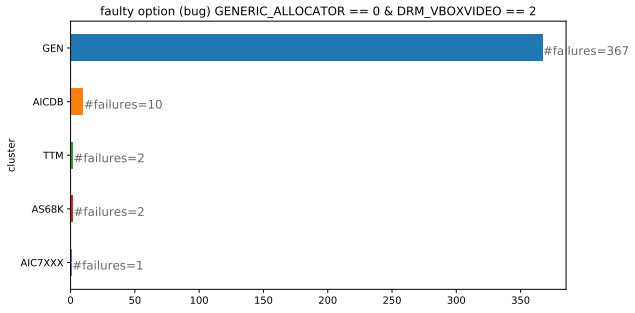


Figure 7: The faulty combination of options GENERIC_ALLOCATOR and DRM_VBOXVIDEO does not necessarily lead to the same failure

in the tree are not geared towards a specific cluster of failure: hence many options are scattered in the tree with misleading interactions. Let us compare the two trees of Figure 4 and Figure 5. In the binary classification setting, the tree is simply unable to point out the option DRM_TTM whereas the rule is immediate and concise for the per cluster classification. For this specific bug, we were unable to find an explanation with binary classification or multi-class classification. We had similar experiences with the understanding of V4L2 and BLACKLIGHT: per cluster classification proved to be a much more effective technique.

Key lesson: limits of pure statistical learning. Owing to the large number of predictors and the scarcity of failures, the sole use of statistical learning has limitations. We analyzed rare failures for which rules extracted from all kinds of classification trees were misleading. However, we made several times the following observation: decision trees chose some options that are statistically accurate yet unrelated to the bug and the failure. Without much insights, we had to look at whether error messages contain relevant information about the potential location of the bug. An open direction is to use heuristics or knowledge to restrict the space of options susceptible to explain a failure: It can help the decision tree choosing more meaningful options.

To what extent bugs' interactions hinder the analysis process? Table II shows that there are configurations with 3 bugs at the same time (*e.g.,* AIC79XX, AIC7XXX, WANXL). There are also several configurations with 2 bugs at the same time. An interesting pair-wise bug interaction is the case VIDEO_MUX and VIDEO_V4L with WANXL_BUILD_FIRMWARE: here we miss an opportunity to observe a failure of VIDEO_MUX and VIDEO_V4L (cluster V4L2 of Table I), since WANXL dominates the bug. In general, we can observe that there is no clear domination of bugs: several clusters may arise given two bugs or three bugs. It further complicates the task of understanding why some failures are observable or not.

Masked failures. A faulty option (or a combination thereof) may arise from several failures. Figure 7 shows that configurations with GENERIC_ALLOCATOR and DRM_VBOXVIDEO do not necessarily lead only to the expected GEN cluster error message. The reason is given by Table II: GENERIC_ALLOCATOR and DRM_VBOXVIDEO can be combined with other faulty options, such as WANXL. In this specific case, the failure AS68K arises (and not GEN). In practical terms, it means that WANXL masks the failure GEN of GENERIC_ALLOCATOR and DRM_VBOXVIDEO. In the worst scenario, there are only configurations with WANXL in the database and developers of TuxML would never notice the issue with GENERIC_ALLOCATOR and DRM_VBOXVIDEO. Fortunately, the failure GEN appears quite frequently (367 times) when GENERIC_ALLOCATOR and DRM_VBOXVIDEO are isolated.

Masked bugs. Given a cluster of errors, an hypothesis is that it is due to an unique configuration bug. Our results refute it: Figure 8 shows that the failure AICDB can be due to 5 possible bugs. In practical terms, it means that developers of TuxML may miss a bug since the very same failure occurs. In the worst scenario, there are only configurations with AICDB failure in the database and developers of TuxML would never be able to differentiate the bug of the 5 faulty configurations.

Key lesson. Beyond these examples, masked failures and masked bugs exist due to many bugs' interactions. The phenomena make the manual analysis process less straightforward. However the effect is manageable: we have numerous configurations in which faulty options are isolated and then related failures can be observed. Statistical learning can then infer that such options are potentially responsible of a failure or a cluster error.

C. (A3) Preventing failures

We aim to address the question: What is the amount of builds needed to accurately prevent failures? To evaluate the model, we used three different metrics (TP is for true positive, TN true negative, *etc.*)

- Accuracy: Percentage of accurately predicted behavior of a build. $accuracy = \frac{TP+TN}{TP+FP+TN+FN}$
- Balanced accuracy: Percentage of accurately predicted behavior of a build, considering the class balance, as failures represent only 4% of the whole dataset. $balanced\_accuracy = \frac{1}{2} \cdot (\frac{TP}{TP+FN} + \frac{TN}{TN+FP})$
- Specificity: Percentage of accurately predicted failures, i.e. prevented failures. $specificity = \frac{TN}{FP+TN}$

We also report on the influence of the size of the training set. Different sets lead to different performances for the trained model, especially when we have a small training set and few dissimilar data. To overcome this problem, we performed a 10-fold cross-validation (CV) for different sizes of a randomly selected training set.

The results are represented in Figure 9 as boxplots. Boxplots gives meaningful insight into a group of values, such as mean (line in the middle), standard deviation (box) and outliers (circles). These boxplots aim to show the performance of the model created, especially the stability over CV of the model. The X-axis represents the training set size, as a percentage of the whole dataset and the Y-axis the score of the metric.

The results show that accuracy is always very good ($\geq$ 90%) due to the class imbalance. Balanced accuracy is much more

degree	faulty_options	nb_failures	clusters
3	[AIC79XX, AIC7XXX, WANXL]	21	[(AS68K, 11), (AICDB, 10)]
3	[AIC79XX, AIC7XXX, GENERIC_ALLOCATOR&DRM_VBOXV...	2	[(AICDB, 2)]
3	[AIC79XX, AIC7XXX, UBSAN_SANITIZE_ALL>.5&INFIN...	4	[(OVERFLOW2, 3), (AICDB, 1)]
2	[VIDEO_MUX&VIDEO_V4L, WANXL]	1	[(AS68K, 1)]
2	[AIC7XXX, WANXL]	63	[(AICDB, 38), (AS68K, 25)]
2	[DRM_VBOXVIDEO&DRM_TTM, GENERIC_ALLOCATOR&DRM_...	2	[(TTM, 2)]
2	[AIC79XX, AIC7XXX]	505	[(AICDB, 462), (AIC7XXX, 29), (AS68K, 11), (OV...
2	[GENERIC_ALLOCATOR&DRM_VBOXVIDEO, WANXL]	2	[(AS68K, 2)]
2	[AIC7XXX, UBSAN_SANITIZE_ALL>.5&INFINIBAND_ADD...	5	[(OVERFLOW2, 4), (AICDB, 1)]
2	[AIC7XXX, GENERIC_ALLOCATOR&DRM_VBOXVIDEO]	8	[(AICDB, 7), (AIC7XXX, 1)]
2	[AIC79XX, UBSAN_SANITIZE_ALL>.5&INFINIBAND_ADD...	7	[(OVERFLOW2, 6), (AICDB, 1)]
2	[AIC79XX, GENERIC_ALLOCATOR&DRM_VBOXVIDEO]	5	[(AICDB, 5)]
2	[AIC79XX, WANXL]	85	[(AS68K, 41), (AICDB, 41), (AIC7XXX, 3)]

Table II: Bugs' interaction (degree is how many faulty options interact; we omit "_BUILD_FIRMWARE" due to space limits)

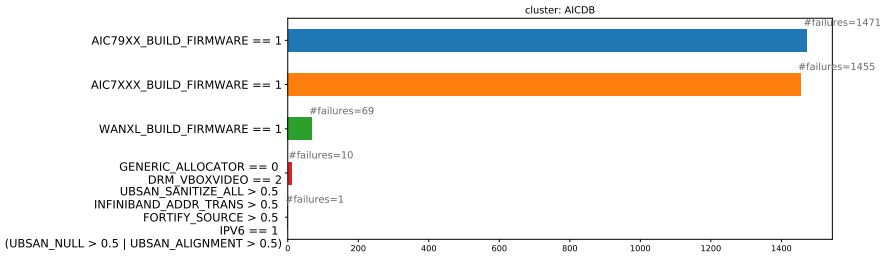
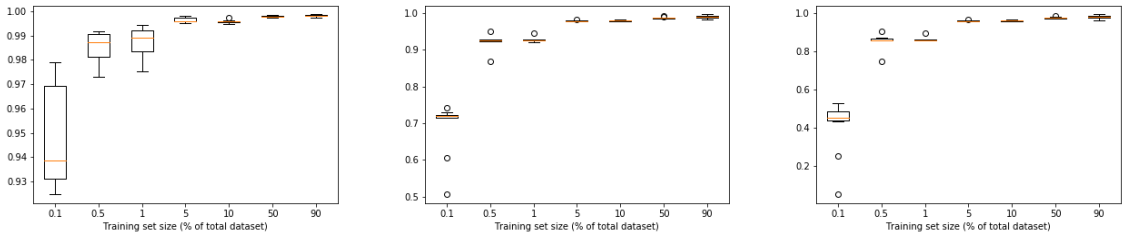


Figure 8: 5 bugs are involved in at least one configuration in which there is the error failure `AICDB` of *e.g.,* Figure 2a



(a) Accuracy (b) Balanced accuracy (c) Specificity

Figure 9: Metrics for decision trees depending on training set size (cross-validation)

insightful for this case. A very low training set size (0.1% of the dataset $\simeq$ 96 configurations) leads to poor performances, but balanced accuracy get quite reliably higher than 90% once the training set is over 0.5% of the dataset ($\simeq$ 480 configurations). At 5% and more, specificity and balanced accuracy are at more than 95% and very stable over CV.

Key lesson. TuxML is able to prevent most of the failures through the identification of faulty options and their interactions. Despite thousands of predictors, we can reach a high accuracy using less than 500 configurations.

VI. Discussions

Comparison with a study about Linux warnings [25] Melo *et al.* compiled 21K valid random Linux kernel configurations for an in-development and a stable version (version 4.1.1). The goal of the study was to quantitatively analyze configuration-dependent *warnings*. One of the major findings is that only 226 did not yield any compilation warning [25]. We follow a similar random strategy, but our goal differs since we focus on build failures rather than warnings. Interestingly, Melo *et al.* report [25]: "we filtered out erroneous configurations (17% of all generated configurations). Errors were dominated by build errors caused by our hardware and installation environment". Our empirical study shows the significance and characterizes the difficulty of the problem. In response, we developed TuxML a learning-based environment capable of building any configuration. Another interesting insight of [25] is as follows: "when building certain firmware drivers in Linux, external proprietary drivers are needed before they can be built. This firmware is not in the kernel code, but must be downloaded from the hardware vendors homepages. In this study, we did not include these firmware drivers [...]". In other words, authors observed similar failures as reported in our study for options WANXL_BUILD_FIRMWARE and PREVENT_FIRMWARE_BUILD (see Section V). TuxML is able to explain such failures and automatically prevent the build of configurations problematic for the environment.

Beyond randconfig? The utility randconfig does *not* provide a perfect uniform distribution over valid configurations [25]. The strategy of randconfig is to randomly enable

or disable according to the order in the Kconfig files. It is thus biased towards features higher up in the tree. The problem of uniform sampling is fundamentally a satisfiability problem. We stick to randconfig for two reasons. To the best of our knowledge, there is no robust and scalable solution capable of comprehensively translating Kconfig files of Linux. Second, uniform sampling either does not scale yet or is not uniform [34]–[37]. The choice of randconfig is obviously a threat to validity since the choice of a better sampler can dramatically change the results. However, we are not there yet. We see randconfig as a *baseline* widely used by the Linux community.

Threats to Validity. Potential threats to the validity of our results are mainly related to the target kernel version and architecture of Linux, the randomness of randconfig, TuxML pre-installed libraries and the chosen clustering technique. Since the kernel version 4.13.3 is a much stable Linux version, it may affect the number of cluster errors and thus influence the per-cluster classification. Although we are aware that the found failures are not automatically transferable to other versions, we are confident about the advantages of using a per-cluster classification against a binary classification.

However, a different cluster environment may influence the ability to install specific libraries which may lead to different results (with either higher TuxML or Linux failures). In order to overcome this threat, we repeated the building process hundreds of times before the experiment. We have invested several months for gathering the TuxML maturity. We have also spent significant effort in denoising data.

For building the cluster, we relied on k-means and an optimization heuristic to find the minimal number of most frequent terms. We were able to cover all configuration fails and build 19 clusters with a set of five key terms. However, other cluster and optimization techniques may be efficiently used and influence the cost of clustering such a big dataset of error messages. Another threat is related to the understanding process of failures and its assessment. We have replicated the reviewing effort and cross-analyzed our results. External experts of Linux could have been involved.

VII. Related Work

A few empirical studies have considered different topics of Linux. Nadi *et al.* [38] mined the Linux repository in the quest of so-called variability anomalies (*e.g.,* mapping code to an invalid configuration). Passos *et al.* [24] analyze the evolution of the configuration space together with the variability implementation of Linux. Abal *et al.* made a qualitative analysis of 42 bugs of the Linux kernel [21], [22]. In [27], authors perform an empirical study of unspecified dependencies in make-based build systems (including Linux). All these works are related to our study. Our research methodology differs: We have built a large corpus of configurations to analyze different aspects of the Linux kernel.

There has been a large body of work that demonstrates the need for configuration-aware testing techniques and proposes methods to sample and prioritize the configuration space [14]–[20], [39], [40]. In [41] a repair process is proposed based on the mining of constraints and testing techniques. In [42], the iTree algorithm is proposed to efficiently coverage configurations using machine learning. In our study, we simply reuse randconfig and did not employ a sophisticated strategy to sample the configuration space. As previously discussed, an exciting research direction is to apply state-of-the-art techniques over Linux but several challenges are ahead.

The *masking effect* has been hypothesized in the context of combinatorial interaction testing [17] and is a source of inspiration for our work. We have defined related concepts (*e.g.,* bugs' interaction) and empirically study the phenomenon, including its impact in practice.

Gazzillo *et al.* [43] challenge the community to locate configurations at a given point of interest. Existing techniques (*e.g.,* Kmax [44]) consider only the build system, and do not inspect configurations within source files. Kuiter *et al.* [45] propose a solution but do not explicitly support Linux "because analyzing such a large system poses new implementation challenges regarding performance optimizations and handling large variability models." Addressing this problem would be very helpful in our context in order to restrict the space of relevant options (see the discussion, page 7).

Zhang and Ernst [46] propose an algorithm to diagnose crashing errors related to software misconfigurations. Their tool works on a single language (Java). We look at more complex configuration spaces. A case study at Google reported a large corpus of builds and build errors mainly focusing on static compilation problems [9]. Beller *et al.* [10] performed an analysis of builds with Travis CI on top of GitHub. One interesting insight is that about 10% of builds show different behavior when different environments are used. It is related to our endeavor to build the "good" distributed environment for testing Linux. Halin *et al.* [47] exhaustively test all possible 26,000+ configurations of an industry-strength, open source configurable software system, JHipster. They found that 35.70% configurations fail and identify the interactions that cause the errors. In the case of Linux, we found fewer failures and seek to identify the faulty combination of options. Halin *et al.* reported the difficulty to engineer a testing infrastructure. We made similar observations and address the problem of continuously handling and preventing failures as early as possible.

VIII. Conclusion

We invested 15K hours of computation to build 95,854 Linux kernel configurations and learnt:

- 3,614 configurations fail (5.83%);
- 3 individual options explain 92% of the failures and are due to configuration-dependent tools that are hard to set up into our infrastructure;
- 16 configuration bugs of Linux explain the rest of the failures: we have been able to automatically find faulty options (and combination thereof) using statistical learning and clustering of error messages;
- bugs' interactions exist and we found configurations with 3 bugs at the same time, masking some failures or bugs;

- it is possible to predict and prevent early failures – being due to the built environment or Linux itself – to avoid unnecessary costly build

Throughout the paper, we report on qualitative and quantitative insights about Linux itself, about the TUXML infrastructure, and about the process of understanding failures within a huge configuration space. Retrospectively and despite our investment, we found relatively few bugs of Linux which calls to address several questions: Is it due to the way we sample? Is it due to the stable version of Linux we chose? Or is it due to the high-quality of Linux, its contributors and its industry-strength, community-based effort?

Acknowledgments. This research was partially funded by the ANR-17-CE25-0010-01 VaryVary project. We would like to thank, in no particular order, Alexis Le Masle, Michaël Picard, Corentin Chédotal, Gwendal Didot, Dorian Dumanget, Antonin Garret, Erwan Le Flem, Pierre Le Luron, Mickaël Lebreton, Fahim Merzouk, Valentin Petit, Julien Royon Chalendard, Cyril Hamon, Paul Saffray, Malo Poles, Luis Thomas, and Alexis Bonnet.

REFERENCES

[1] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Computing Surveys*, vol. 47, no. 1, pp. 6:1–6:45, 2014. 1

[2] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin, "Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1069–1089, aug 2013. 1

[3] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Exploring variability-aware execution for testing plugin-based web applications," in *Proceedings of the 36th International Conference on Software Engineering - ICSE '14*. ACM, 2014, pp. 907–918. 1

[4] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. d'Amorim, "Splat: lightweight dynamic analysis for reducing combinatorics in testing configurable systems - esec/fse '13," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 257–267. 1

[5] C. H. P. Kim, D. S. Batory, and S. Khurshid, "Reducing combinatorics in testing product lines," in *Proceedings of the tenth international conference on Aspect-oriented software development*, ser. AOSD '11. ACM, 2011, pp. 57–68. 1

[6] Y. Chen, F. Wu, K. Yu, L. Zhang, Y. Chen, Y. Yang, and J. Mao, "Instant bug testing service for linux kernel," in *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, Nov 2013, pp. 1860–1865. 1, 7

[7] 0day infrastructure, "https://01.org/lkp/documentation/0-day-test-service." 1, 7

[8] M. Greiler, A. van Deursen, and M. A. Storey, "Test confessions: A study of testing practices for plug-in systems," in *34th International Conference on Software Engineering (ICSE)*. ACM, 2012, pp. 244–254. 1

[9] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' Build Errors: A Case Study (at Google)," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 724–734. 1, 10

[10] M. Beller, G. Gousios, and A. Zaidman, "Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub," in *Proceedings of the 14th International Conference on Mining Software Repositories*, ser. MSR '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 356–367. 1, 10

[11] N. Kerzazi, F. Khomh, and B. Adams, "Why do automated builds break? an empirical study," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 41–50. [Online]. Available: https://doi.org/10.1109/ICSME.2014.26 1

[12] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 159–172. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043572 1

[13] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo, "Software configuration engineering in practice: Interviews, survey, and systematic literature review," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018. 1

[14] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry, "Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack," *Empirical Software Engineering*, Jul 2018. [Online]. Available: https://doi.org/10.1007/s10664-018-9635-4 2, 10

[15] M. Sayagh, N. Kerzazi, and B. Adams, "On cross-stack configuration errors," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 255–265. [Online]. Available: https://doi.org/10.1109/ICSE.2017.31 2, 10

[16] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 643–654. [Online]. Available: https://doi.org/10.1145/2884781.2884793 2, 10

[17] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter, "Feedback driven adaptive combinatorial testing," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 243–253. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001450 2, 10

[18] X. Niu, n. changhai, H. K. N. Leung, Y. Lei, X. Wang, J. Xu, and Y. Wang, "An interleaving approach to combinatorial testing and failure-inducing interaction identification," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018. 2, 10

[19] D. Jin, X. Qu, M. B. Cohen, and B. Robinson, "Configurations everywhere: Implications for testing and debugging in practice," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 215–224. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591191 2, 10

[20] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 20–34, 2006. 2, 10

[21] I. Abal, J. Melo, x. Stănciulescu, C. Brabrand, M. Ribeiro, and A. Wasowski, "Variability bugs in highly configurable systems: A qualitative analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 3, pp. 10:1–10:34, Jan. 2018. [Online]. Available: http://doi.acm.org/10.1145/3149119 2, 10

[22] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: a qualitative analysis," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 421–432. [Online]. Available: https://doi.org/10.1145/2642937.2642990 2, 10

[23] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Where do configuration constraints stem from? an extraction approach and an empirical study," *IEEE Trans. Software Eng.* 2

[24] L. Passos, R. Queiroz, M. Mukelabai, T. Berger, S. Apel, K. Czarnecki, and J. Padilla, "A study of feature scattering in the linux kernel," *IEEE Transactions on Software Engineering*, 2018. 2, 10

[25] J. Melo, E. Flesborg, C. Brabrand, and A. Wasowski, "A quantitative analysis of variability warnings in linux," in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*, ser. VaMoS '16. ACM, 2016, pp. 3–8. 2, 6, 9

[26] N. Dintzner, A. van Deursen, and M. Pinzger, "Analysing the linux kernel feature model changes using fmdiff," *Software and System Modeling*, vol. 16, no. 1, pp. 55–76, 2017. [Online]. Available: https://doi.org/10.1007/s10270-015-0472-2 2

[27] C. Bezemer, S. McIntosh, B. Adams, D. M. Germán, and A. E. Hassan, "An empirical study of unspecified dependencies in make-based build systems," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3117–3148, 2017. [Online]. Available: https://doi.org/10.1007/s10664-017-9510-8 2, 10

[28] Supplementary material (Web page), "https://github.com/tuxml/compilation-analysis." 2
[29] K.-J. Stol and B. Fitzgerald, "The abc of software engineering research," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, pp. 11:1–11:51, Sep. 2018. [Online]. Available: http://doi.acm.org/10.1145/3241743 2
[30] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *34th International Conference on Software Engineering*, 06/2012 2012. 2
[31] L. Passos, R. Queiroz, M. Mukelabai, T. Berger, S. Apel, K. Czarnecki, and J. Padilla, "A study of feature scattering in the linux kernel," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018. 2
[32] A. P. Mathur, *Foundations of software testing*. India: Pearson Education, 2008. 2
[33] I. C. Society, P. Bourque, and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*, 3rd ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014. 2
[34] Q. Plazar, M. Acher, G. Perrouin, X. Devroey, and M. Cordy, "Uniform sampling of sat solutions for configurable systems: Are we there yet?" in *ICST 2019 - 12th International Conference on Software Testing, Verification, and Validation*, Xian, China, Apr. 2019, pp. 1–12. [Online]. Available: https://hal.inria.fr/hal-01991857 10
[35] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable and nearly uniform generator of sat witnesses," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 608–623. 10
[36] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "On parallel scalable uniform SAT witness generation," in *Tools and Algorithms for the Construction and Analysis of Systems TACAS'15 2015, London, UK, April 11-18, 2015. Proceedings*, 2015, pp. 304–319. 10
[37] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, "Efficient sampling of SAT solutions for testing," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 549–559. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180248 10
[38] S. Nadi, C. Dietrich, R. Tartler, R. C. Holt, and D. Lohmann, "Linux variability anomalies: What causes them and how do they get fixed?" in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 111–120. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487112 10
[39] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Computing Surveys*, 2014. 10
[40] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon, "Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 650–670, 2014. 10
[41] A. Gargantini, J. Petke, and M. Radavelli, "Combinatorial interaction testing for automated constraint repair," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2017, pp. 239–248. 10
[42] C. Song, A. Porter, and J. S. Foster, "itree: Efficiently discovering high-coverage configurations using interaction trees," *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 251–265, March 2014. 10
[43] P. Gazzillo, U. Koc, T. Nguyen, and S. Wei, "Localizing configurations in highly-configurable systems," in *Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018*, 2018, pp. 269–273. [Online]. Available: https://doi.org/10.1145/3233027.3236404 10
[44] P. Gazzillo, "Kmax: finding all configurations of kbuild makefiles statically," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 279–290. [Online]. Available: https://doi.org/10.1145/3106237.3106283 10
[45] E. Kuiter, S. Krieter, J. Krüger, K. Ludwig, T. Leich, and G. Saake, "Pclocator: A tool suite to automatically identify configurations for code locations," in *Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 1*, ser. SPLC '18. New York, NY, USA: ACM, 2018, pp. 284–288. [Online]. Available: http://doi.acm.org/10.1145/3233027.3236399 10
[46] S. Zhang and M. D. Ernst, "Automated diagnosis of software configuration errors," in *2013 35th International Conference on Software Engineering (ICSE)*, May 2013, pp. 312–321. 10
[47] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and P. Heymans, "Yo variability! JHipster: A playground for web-apps analyses," in *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*, ser. VAMOS '17. New York, NY, USA: ACM, 2017, pp. 44–51. 10